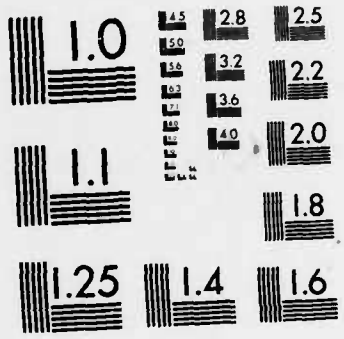
AD-A134 857 FEASIBILITY ASSESSMENT OF JOVIAL TO ADA TRANSLATION(U) 1/1
AIR FORCE WRIGHT AERONAUTICAL LABS WRIGHT-PATTERSON AFB
OH D H EHRENFRIED AUG 83 AFWAL-TR-83-1058

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFWAL-TR-83-1058

AD-A134 857

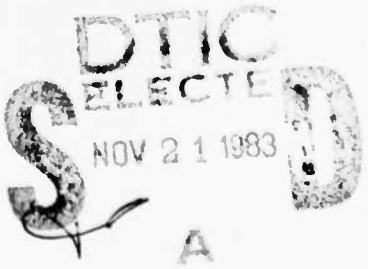
FEASIBILITY ASSESSMENT OF JOVIAL TO ADA TRANSLATION

Daniel H. Ehrenfried, 1Lt, USAF
AFWAL/AAAF-2
Wright-Patterson AFB, Ohio 45433

August 1983

Final Report for Period 1 June 1982 to 29 August 1982

Approved for Public Release; Distribution Unlimited.



AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433

DTIC FILE COPY

83 11 21 008

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

DANIEL H. EHRENFRIED, 1Lt, USAF
Support Software Group
Avionics Laboratory

ISRAEL I. CARO, Maj, USAF
Chief, Support Systems Branch
Avionics Laboratory

FOR THE COMMANDER

FRANK A. SCARPINO, Acting Chief
System Avionics Division
Avionics Laboratory

"If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAF, W-PAFB, OH 45433 to help us maintain a current mailing list."

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFWAL-TR-83-1058	2. GOVT ACCESSION NO. A134857	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* FEASIBILITY ASSESSMENT OF JOVIAL TO ADA TRANSLATION		5. TYPE OF REPORT & PERIOD COVERED Technical Paper June 1982 - August 1982 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Daniel H. Ehrenfried, 1Lt, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Avionics Laboratory (AFWAL/AAAF-2) Air Force Wright Aeronautical Laboratories (AFSC) Wright-Patterson AFB, OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 20030304
11. CONTROLLING OFFICE NAME AND ADDRESS Avionics Laboratory (AFWAL/AAAF-2) Air Force Wright Aeronautical Laboratories (AFSC) Wright-Patterson AFB, OH 45433		12. REPORT DATE August 1983 13. NUMBER OF PAGES 48
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15e. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Ada
JOVIAL
Automatic Translation

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The Ada program is part of a DoD policy change towards incrementally reducing the number of languages in use by the DoD from many to only a few and then eventually to just one; Ada. Currently, MIL-STD-1589B (JOVIAL - J73) is the Air Force standard language for use in the embedded application domain. One approach towards an earlier transition of all software written in Ada would be the development of an automatic J73 to Ada translation system. With the translation of all J73 software into Ada, J73 software development

(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Item 20 Continued.

systems could be phased out of use, the cost of maintaining the J73 system could be recovered, and programmers would be freed earlier for their eventual transition to Ada. This paper will examine the feasibility and cost effectiveness of developing a J73 to Ada translation system.

FOREWORD

This report discusses the technical feasibility of performing automatic translation of programs written in JOVIAL to those written in Ada.

The work reported herein was performed during the period 1 June 1982 to 19 August 1982 by Lieutenant Daniel Ehrenfried for the Air Force Wright Aeronautical Laboratories (AFWAL) for Project 2003, Task 03, Work Unit 04.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

TABLE OF CONTENTS

SECTION		PAGE
I	INTRODUCTION	1
II	REQUIREMENTS	3
	1. Source-to-Source Translation	3
	a. Execution Equivalence	3
	(1) Efficiency	4
	b. Source Code Quality	5
	(1) Maintainability and Reliability	5
	(2) Robustness	6
	2. A J73 to Ada Translation System	7
	a. Scope	7
	b. Semantic Equivalence	8
	c. Efficiency	9
	d. Source Code Quality	10
	3. Requirements Summary	10
III	ANALYSIS	11
	1. Automatic J73 to Ada Translation	11
	a. Global Concepts	11
	(1) The Complete Program	11
	(2) Modules	11
	(a) Compool Modules	11
	(3) Scope of Names	12
	(4) Implementation Parameters	12
	b. Declarations	13
	(1) Data Declarations	13
	(a) Item Declarations	13
	(1) Integer Type Descriptions	13
	(2) Floating Type Descriptions	14
	(3) Fixed Type Descriptions	15
	(4) Bit type Descriptions	15
	(5) Character Type Descriptions	15
	(6) Status Type Descriptions	16
	(7) Pointer Type Descriptions	16

PREVIOUS PAGE IS BLANK

TABLE OF CONTENTS (Cont'd)

SECTION			PAGE
III		(b) Table Declarations	17
		(1) Table Dimension Lists	17
		(2) Table Structure	18
		(3) Ordinary Table Entries	18
		(4) Specified Table Entries	18
		(c) Constant Declarations	18
		(d) Block Declarations	18
		(e) Allocation of Data Objects	19
		(f) Initialization of Data Objects	19
	(2)	Type Declarations	20
	(3)	Statement Name Declarations	20
	(4)	Define Declarations	20
	(5)	External Declarations	20
	(6)	Overlay Declarations	21
	c.	Procedures and Functions	21
	(1)	Procedures	21
	(2)	Functions	21
	(3)	Parameters of Procedures and Functions	22
	(4)	Inline Procedures and Functions	23
	(5)	Machine Specific Procedures and Functions	23
	d.	Statements	23
	(1)	Assignment Statements	23
	(2)	Loop Statements	24
	(3)	IF Statements	24
	(4)	CASE Statements	24
	(5)	Procedure Call Statements	25
	(6)	RETURN Statements	25
	(7)	GOTO Statements	25
	(8)	EXIT Statements	26
	(9)	STOP Statements	26
	(10)	ABORT Statements	26

TABLE OF CONTENTS (Cont'd)

SECTION		PAGE
III	e. Formulas	26
	(1) Numeric Formulas	26
	(a) Integer Formulas	26
	(2) Bit Formulas	27
	(a) Relational Expressions	27
	(b) Boolean Formulas	27
	f. Data References	27
	(1) Variables	27
	(2) Named Constants	28
	(3) Function Calls	28
	g. Type Matching and Conversions	29
	h. Basic Elements	30
	(1) Characters	30
	(2) Symbols	30
	(a) Names	30
	(b) Reserved Words	31
	(3) Literals	31
	(a) Numeric Literals	31
	(b) Bit Literals	31
	(c) Boolean Literals	32
	(d) Character Literals	32
	(4) Comments	32
	i. Directives	32
	2. Summary of Untranslatable Features	33
	3. Percentage Translatable	35
	4. Summary of Unused Ada Constructs	35
	5. Cost Effectiveness	36
	a. Translation System Development Costs	36
	b. Code Translation Costs	37
	c. J73 versus Ada	37
IV	CONCLUSIONS AND RECOMMENDATIONS	38

LIST OF ILLUSTRATION

FIGURE		PAGE
1	J73 - Ada Definition Conflicts	34

SECTION I

INTRODUCTION

An initial impetus for standardization on a single High Order Language (HOL) for Department of Defense software applications was to reduce the overabundance of languages requiring support by the DOD. Each language requires a pool of people knowledgeable in the details of that particular language, a compiler, and a set of related support tools in order to write and maintain software in that language. Often, people and resources are not easily interchangeable between different language systems. This redundancy illuminated the need to consolidate to a smaller set of languages that could still satisfy DOD requirements in all application areas.

Currently, MIL-STD-1589B (JOVIAL - J73) is the Air Force standard language for use in the Embedded Computer System (ECS) domain. Software written today in J73 will probably last throughout the lifetime of the weapon system in which it is used. History shows that this life span is normally between 5 and 10 years, and often is more than 15. A J73 language system (compiler and support tools) and the necessary complement of trained personnel will also be required throughout this life cycle.

The Ada program is part of a DOD policy change towards incrementally reducing the number of languages in use from many to only a few, and then eventually to just one -- Ada. One approach towards an earlier transition from all software written in J73 to software written in Ada would be the development of an automatic J73 to Ada translation system. With the translation of all J73 software into Ada, the J73 language system could be phased out of use, the cost of maintaining the J73 system could be recovered, and programmers would be freed earlier for their eventual transition to Ada. The question is whether this is a rational step towards standardization on a single HOL.

Three issues must be addressed before this approach should be adopted by the Air Force. First, the feasibility of an automatic translation system must be demonstrated. It must be proven that a sufficient percentage of the J73 language can be correctly translated into Ada. Second,

the impact of translation upon the quality of the resulting software must be measured. A translation system must not only produce a correct translation but also maintain the quality of the resulting software including its maintainability, reliability, and robustness. Within the highly constrained environment of real-time embedded software, it is critical that efficiency be preserved as well. Finally, the cost effectiveness of this approach must be shown. The cost of performing the translation must be weighed against the expected savings from an early transition to Ada.

This report will first outline a set of general requirements for generic source-to-source translation. These requirements will then be refined to incorporate specific characteristics of J73, Ada, the embedded applications environment within which actual J73 software would be translated into Ada, and the state of the art in translation technology. A technical analysis of the feasibility of such a translation system will then be given. Finally, a set of conclusions will be drawn and a list of recommendations for the use of this technology presented.

SECTION II

REQUIREMENTS

Section II.1 will define a set of general requirements for performing source-to-source translation at the HOL level. Section II.2 will relate these requirements to an actual J73 to Ada translation system.

1. SOURCE-TO-SOURCE TRANSLATION

Any translation system must preserve the characteristics of the original program in two major ways: 1) execution equivalency including functionality and efficiency and 2) source code quality. The following discussions are intended to present a minimum set of requirements at the highest, most general level.

a. Execution Equivalence

The original program and the resulting translation must be equivalent to the largest extent possible. Exact equivalence might be defined as two source code modules which, when compiled, produce the exact same load module for a given target. Even if this were possible to attain, exact equivalence is certainly not necessary. The critical measure is that the two pieces of software are functionally equivalent; they perform the same task.

Thus, at a minimum, any functional requirements placed upon the original code must be preserved during translation. The resulting translation must produce the same "effect" on the outside world as the original. Other more specific restrictions might include:

1. External inputs should be interpreted and stored in the same manner.
2. For a given set of inputs, outputs should emanate with same values and in the same order.
3. Critical timing dependencies within the original program must be met by the resulting translation.

(1) Efficiency

The discussion of efficiency often reduces to a question of tradeoffs between available resources and time. In programming terms, one major resource is storage space and time is measured in execution cycles. Space can be optimized by packing data into the smallest representation possible. Unfortunately, additional effort must then be exerted (and time expended) to extract the data when it is required, and replace it after it has been modified. Through the use of data redundancy, processing time can, in most cases, be reduced. All applications must strike an appropriate balance between space and time to fit their underlying hardware resources and meet any timing requirements.

Programs must not only be translated correctly, but also preserve the efficiency characteristics of the original program. Again, exact equivalence between two programs written in different HOLs would require that the same number of machine instructions be used to implement all functional aspects, and that the same amount of storage be used for any accompanying data. This goal is just as unattainable as exact functional equivalence and even more unnecessary. A more reasonable requirement might be to restrict the translation to be, on average, no less efficient than the original. The overall size of the code and execution speed of the translation should not exceed that of the original in any considerable way. Minor local aberrations may in some cases be tolerable, but approximate global parity must be preserved and local deviations must not be too large.

Translation requirements should certainly not prohibit increases in efficiency when they can be realized. In most cases it would be desirable to have increased efficiency in the resulting translation unless, by doing so, a timing restriction is violated. In practice, however, efficiency increases are very difficult to achieve. The implementation of a particular function in software for a specific target hardware will require a certain minimum amount of data storage and instructions. Two compilers of equal quality, each employing similar optimization techniques, will both approach this minimum to approximately the same extent.

b. Source Code Quality

If the software is expected to have a reasonably long life cycle, it must be of high quality. Software systems are constantly being modified and updated to fix newly detected errors and reflect changing requirements. Quality software can make this process easier and more cost effective.

Execution characteristics are not the only measure by which the quality of software is judged. Maintainability and reliability are critical metrics of quality. It should be as readable, easily understandable, and embrace the style and intent of the language in which it is coded. Translations should also result in robust implementations, using to the fullest extent possible the power of the target HOL. An equivalent or greater level of these qualities should be present in the software resulting from translation.

(1) Maintainability and Reliability

"Maintenance" is a deceptive term when applied to software. It does not imply that one must apply constant tinkering to maintain a constant level of functioning as one might a piece of machinery. Except for the possibility of hardware error, programs should by definition execute in the same manner each time they are invoked with the same inputs. This term actually refers to the fixing of bugs in software that deviate from the original set of functional requirements or the modification of a program to reflect a new set. Several qualities make the maintenance of software easier to perform. Software should be well structured with separate modules for separate functions. It should exhibit clear data flow between modules and clear flow of control within. It should be well documented and commented in a manner that promotes the understanding of its intent. The specification or functional interface of a subprogram and data should be separate from its implementation.

"Reliability" is a related term that is probably misapplied as well. The definition used here is the quality of a program to isolate the effects of inevitable programming bugs. Two aspects of an HOL and its programming environment can improve the reliability of software.

The first is the automatic detection of bugs. Through exact specification (typing) and redundant information about intent (declarations) many bugs can be found at compile time, and corrected immediately. A second aspect is a program's ability to localize the effect of modifications to a program. Subtle dependencies between two sections of software within one program can allow errors to occur in one section when changes are made to the other. (Global flags are a good example of such dependencies.) Dependencies other than those explicitly placed in the software in a clear manner should be avoided whenever possible. Third, access to the internal definition of data and functions should be restricted only to parts of the program that require it. This is often called Information Hiding. Multiple access paths to a single data object, commonly called aliasing, should also be avoided. Modularization and the grouping of similar functions together contribute to reliable code.

A translation system should preserve or improve the level of these qualities in the resulting program. This is often very difficult to do. The quality of software is frequently inherent in the design of the software and the features of the language used in the implementation. The original software may even violate some of the quality standards given above. To improve maintainability, one must either redesign the module or find a way to use features of the target HOL that are designed to improve software quality. Redesign of software is currently considered beyond the scope of an automatic translation system. We can reasonably attempt, therefore, to preserve only those qualities that are present in the original software.

(2) Robustness

Robustness is a measure of how well a program utilizes the features of the language in which it is written. Software written in a particular HOL should take advantage of its powerful features whenever possible. This not only improves efficiency but also provides a clearer representation of what the program is intended to do. Features are placed in a language for a reason and should be used whenever the intent of the feature is applicable to the requirement at hand. A translation system should attempt to provide the highest possible utilization of the target HOL.

Often, there is no one-to-one correspondence between the features of the target and source HOLs. Features in the source HOL which do not have an equivalent feature in the target HOL will have to be implemented with a combination of lower level features. The reverse is also true. Features in the target HOL may not be used because there is no corresponding feature in the source HOL from which it can be translated. Thus, higher level features in the target HOL may be only partially used or not at all.

2. A J73 TO ADA TRANSLATION SYSTEM

This section will now consider translation system requirements when the source HOL is JOVIAL / J73, Ada is the target HOL, and the type of software to be translated is real-time embedded software.

a. Scope

A compiler is an example of a system that translates programs written in a high order language into equivalent programs "written" in machine language. A source-to-source translation system would have to perform many of the same "front end" analysis functions as a compiler for the same source HOL, the difference lying in the level of the target language. The capabilities of today's compiler technology can therefore be used as a baseline for analyzing the limits of a source-to-source translation system.

From the outset, we must remind ourselves of the limited capacity to which machines (or the programs that run on them) can understand programs written in one language and translate them into another. An ideal system would accept any legal J73 program and return the Ada equivalent. In practice, however, translation of J73 to Ada will require a mixture of both automatic translation and human augmentation.

Two things can be stated with certainty:

1) All J73 programs have a functionally equivalent implementation in Ada.

J73 and Ada are alike in many ways. Although difficult to prove, neither can implement a function that the other cannot duplicate.

Turning Equivalence should guarantee that a skilled programmer can design and code an algorithm in both languages that is functionally equivalent. (Ada was designed to be an improvement over J73. As such, programs written in Ada should be of higher quality. Improvements were primarily made in the features for structuring programs and not to the functional capability of the language. Ada did formalize several areas such as error handling, multi-process control, and I/O)

2) The translation from J73 to Ada cannot be a 100% automated process.

J73 and Ada are also dissimiliar in many ways. Several J73 features have no corresponding feature in Ada. When a particular feature cannot be translated directly, some alternative must be found. A translation system may not be able to understand the meaning of a whole program well enough to find an alternative combination of Ada features that provide the same effect. A human will have to step in and redsign sections in Ada, then integrate them with the rest of the program. The extent to which this must be done is the critical measure of a translation system's viability.

b. Semantic Equivalence

Section II.1.a stated the requirement that the original program and resulting translation be "functionally equivalent". Unfortunately, today's compilers can not "understand" at this high a level. They can recognize and understand most constructs at the statement level only. Thus, "functional equivalence" must also be applied at this level. A J73 to Ada translation system must have an equivalent Ada statement or statements for every J73 statement or mark it as untranslatable. (Both languages contain definitions of more than just "statements." Here, reference to a "statement" is intended to mean any separately defined construct of the language, including those constructs not formally defined as statements.)

In order to make a comparison between two similar statements from two different languages, and arrive at a judgement of their equivalence or lack thereof, a precise definition must be available for

both the "source" and "target" statements. Unfortunately, MIL-STD-1589B contains many constructs for which the definition is incomplete or ambiguous. This allows the compiler writer the freedom to interpret the definition to mean either what is most logical to him or easist to implement. J73 also defines several parts of the language to be "implementation dependent," again allowing the compiler writer the freedom of choice. The same is true for the definition of Ada, but to a much more limited extent. Every attempt was made during the language definition to provide the most complete definition possible, to remove any abiguities, and to isolate machine dependencies.

Incomplete, ambiguous, and implementation dependent definitions have dire consequences for general purpose J73 to Ada translation systems. In order to have the widest possible application, such a translation system would have to be flexible enough to adapt to the various interpretations that particular J73 and Ada implementations have adopted. Another option might be to have several translation systems, each tailored to a particular compiler pair, although the economics of such a solution would probably be prohibitive. Areas where an ambiguous or incomplete definition effect the ability to perform correct translation will be pointed out in Section III.1.

c. Efficiency

Real-time software systems are bound by very stringent efficiency requirements. Memory is usually limited and the computing power is seldom adequate to execute every desirable function. Software often has to be "shoehorned" into memory with little if any space to spare. A translating system must therefore minimize its impact on both time and space. There are several pitfalls awaiting our attempts to translate J73 to Ada. Numeric accuracy and data representation are defined differently in J73 and Ada. The translation of some features may induce some additional overhead. Differences between implementation dependent parameters and options are especially troublesome. Each compiler, both the original J73 compiler and the Ada compiler for the translated program, will make space/time tradeoff decisions in a different manner.

d. Source Code Quality

Real-time software is constantly being updated and modified. Source code quality is essential to performing these upgrades in a cost effective manner. A translation system must insure a high degree of readability and limit the amount of underlying dependencies that result in unreliable code.

3. REQUIREMENTS SUMMARY

1. The semantic equivalence between J73 statements or blocks of statements and Ada must be guaranteed.

2. Induced processing overhead must be minimized. The exact toleration threshold is application dependent.

3. Data storage requirements must remain approximately equivalent. This threshold is also application dependent.

4. The translation must produce readable code. It should be well structured in the style and intent of Ada.

5. The resulting code should be free of subtle underlying dependencies.

6. The translation system should utilize the features of Ada to the largest extent possible.

SECTION III

ANALYSIS

1. AUTOMATIC J73 TO ADA TRANSLATION

This section will mirror the structure of the MIL-STD-1589B definition of the J73 language. Chapter titles in the Standard match subsection titles here. All comments relating to the translation of J73 constructs will appear in the appropriate section. Any violations of the requirements given in II.3 will be so noted. All references to the Ada Language Reference Manual (LRM) refer to the July 1982 version of that document.

a. Global Concepts

(1) The Complete Program

The concepts of modules, complete programs, and main program modules parallel those of Section 10.1 in the Ada Language Reference Manual (LRM).

(2) Modules

(a) Compool Modules

Compools can be translated into Ada packages with some minor caveats. Ada requires a separate package specification and body when subprograms are included. Separate specification and body modules can be fabricated by a translator from the locally available information. This simply complicates the work required of a translator. Most compool directives (also discussed in Section III.1.i) can be mapped into the Ada WITH CLAUSE. J73 REF and DEF specifications are defined independently of compools and will be discussed in Section III.1.b.(5).

The Ada package is actually much more powerful than the J73 Compool. The following list of unreachable capabilities indicates a conflict with Requirement 6 of Section II.3.

- Packages can contain variables. Compools can declare variables, but only with the external DEF construct. Problems with this are discussed in Section III.1.b.(5).

- Compools have nothing corresponding to private or limited private types.

- Package bodies may contain internal declarations not visible outside the package for use in the internal implementation of the package specification. It must be assumed that all declarations within a compool are intended for external use.

- Package bodies may also contain an executable part similar to the BEGIN ... END of a subprogram definition. Compools have no corresponding capability.

(3) Scope of Names

No conflicts with the Ada visibility rules given in section 8 of the Ada LRM appear in this section. External names will be discussed in Section III.1.b.(5).

Two conflicts with Requirement 6 appear:

- J73 prohibits two names within the same scope to have the same spelling. Ada allows for the overloading of subprogram names and enumeration literals.

- Name conflicts and overloading ambiguities are avoided in Ada through the RENAMES facility. This is not available in J73.

(4) Implementation Parameters

The J73 LRM contains the following statement: "The machine on which a J73 program runs contains an array of memory cells." Ada does not make this specific a statement about the hardware on which it will run.

This difference contains several implications. The most obvious one appears in this section, namely the existence of implementation parameters relating to linear memory. Programs which refer to these parameters will require the same value when translated into Ada. Each of these constants will have to be encapsulated in a package similar to the SYSTEM package for use throughout the program. Other implementation parameters are J73 dependent, i.e., MAXTABLESIZE. Tables are obviously

not in Ada. The name "MAXTABLESIZE" would make no sense in an Ada program. Parameters such as this will probably have to be hand translated. Other numeric parameters may have to be translated by hand as well.

b. Declarations

(1) Data Declarations

Data manipulation, especially of numerical data, is the core of any programming language. The majority of the constructs in J73 and Ada are for structuring programs and for managing the flow of control between functional subsections. The bottom line, however, is the manipulation of hard data, the handling of input from the outside world. The primary issues involved here are the representation of data and its precision, range constraints, and allocation permanence. Embedded systems require a strict definition of precision in order to maintain the accuracy of numerical computations. They may also require precise control over the legal range of variables. These issues and their translation from their meaning in J73 to that in Ada are discussed in the following sections.

(a) Item Declarations

(1) Integer Type Descriptions

J73 and Ada differ in the manner in which they define the range of integers. Ada allows the range to be arbitrary (within the bounds of the SYSTEM INTEGER'RANGE) and be expressed in decimal or as a based number. J73 requires that "... the minimum number of bits required to hold the maximum value of the integer (excluding the sign, if any)..." be given in the ITEM declaration of an integer. While this does allow for the implication of a range constraint, it offers them only with limits of powers of two. For example, the declaration:

```
ITEM X U 4;
```

declares the unsigned integer X with 4 bits to hold its values. This implies a range of 0 to 15.

There are several problems with the J73 definition provided. It does not define what happens when an attempt is made to assign to an integer ITEM with a value larger than it is allowed to hold. Does rounding or truncation occur? Are the high order bits masked, performing something like modulo arithmetic? Is the size constraint ignored resulting in no effect. Does the execution of the program halt? MIL-STD-15898 simply does not say.

Ada contains the concept of a range constraint. An exception, CONSTRAINT_ERROR, will be raised if an attempt is made to assign a value outside the declared range. J73 has no concept of exceptions, exception handlers, or even of error conditions. One approach towards avoiding CONSTRAINT_ERROR exceptions would involve translating all integer definitions into the SYSTEM defined integer type ignoring any <integer-size> attributes. There are two problems with this. One, an exception cannot be avoided entirely if an attempt is made to assign a value outside INTEGER'RANGE. And two, it is hard to justify ignoring this attribute when the original programmer took the time to specify it, and must have done so for a reason.

The J73 <round-or-truncate> attribute is also troublesome. Either rounding or truncation is invoked during type conversion in J73 (specified by the ITEM declaration of the target variable). The exact algorithms for truncation and rounding are not specified and so must be assumed to be implementation dependent. The J73 manual statement "If the [<round-or-truncate>] attribute is omitted, truncation in an implementation-dependent manner will occur," further muddies the water.

Ada allows for explicit type conversions between "closely related numeric types." No mention is made in the Ada manual of any rounding or truncation except for: "The conversion of a real value to an integer type rounds to the nearest integer; if the operand is half way between two integers (within the accuracy of the real subtype), rounding may be either up or down."

(2) Floating Type Descriptions

The definition of J73 floating point numerics have many of the same problems as that of integers. The <precision> field

again refers to the number of bits needed to represent the mantissa. It offers no ability to specify a range constraint. Ada requires precision to be defined as the number of decimal digits for the mantissa and allows for a range constraint. Anomalies between the representation of the mantissa as decimal digits and binary bits may cause problems.

The definition of rounding and truncation is missing. If the attribute is omitted, truncation in an implementation dependent manner will again occur.

(3) Fixed Type Descriptions

Precision and range are again specified in numbers of bits. J73 fixed type declarations contain two attribute fields. The <scale-specifier> indicates the number of bits to the left of the decimal point including the sign bit. It is unclear whether this implies a range constraint similar to that for integers. The <fraction specifier> indicates the number of bits to the right of the decimal point. Again, Ada differs in its specification semantics for this data type. Ada allows specification of a delta and a range constraint. It is unclear whether these two definitions are compatible in all cases. A definition for rounding and truncation is implementation dependent and not provided in the manual.

(4) Bit Type Descriptions

Ada does not provide a bit string type directly. The Ada LRM does make special reference to objects declared as:

```
type BIT_VECTOR is array (NATURAL range <>) of BOOLEAN
```

Since the BIT_VECTOR type is defined as an array of BOOLEANS, use of this type will be governed by all rules relating to arrays. The functions "and", "or", and "xor" which operate directly on objects of this type are provided and are presumably optimized.

(5) Character Type Descriptions

The type STRING is also constrained by its definition as an array.

(6) Status Type Descriptions

J73 Status types translate fairly easily into Ada enumeration types with some minor reformatting. Two problems are apparent:

- Ada does not have an equivalent of the <status-size> attribute.

- In Ada, enumeration types must be named types. Objects cannot be directly declared as enumeration types as with arrays, etc. Thus, a type declaration and type name must be generated when a status declaration is translated into an Ada enumeration type. The problems associated with generating an appropriate name will be discussed in Section III.1.h.(2).(a).

(7) Pointer Type Descriptions

J73 pointers appear to be equivalent to Ada access types. In fact, they are not.

Access types are included in Ada for two reasons. Their primary purpose is as the mechanism for naming dynamically created objects. Static objects are given a name reference at declaration time. Dynamically created objects are given an internal name by which to reference them. Access types hold these name values. Access values are typed in that they can only hold references to objects of one type. Access types also provide a convenient way to implement directed graph structures.

J73 pointers differ in the following respects. A minor difference is that pointers can be untyped in J73. Untyped pointers will not translate into Ada. The major difference is that pointers are actually defined to be the address of the object pointed to. The functions LOC and NEXT move pointers around the address space allowing access to the internal structure of all data objects. Pointers can also be converted into integers and bit strings. This allows manipulation with integer and bit string operators. The values can then be converted back into a pointer. In this way, all data (and possibly even instructions) is

exposed to meddling from anywhere in the program. Ada was designed specifically to prevent programmers from accessing data in this manner. Pointers, therefore, can not be translated into access types. All code involving pointer types will have to be hand translated. This is a very serious violation of requirement 1.

Note: Since J73 does not allow dynamic allocation (other than block entry), access types will not be used by a translation system.

(b) Table Declarations

There is no directly parallel structure for J73 TABLES in Ada. Table-like structures can be composed with an array of records. This works fairly well with several small problems. Most of the incompatibilities occur with the several special case rules connected with TABLES.

Ada requires record types to have a name in a similar manner to enumeration types. Objects cannot be directly declared as a record. They can be declared only as a record type declared elsewhere. This requires a name to be generated for the record type to match the internal structure of the table. Name generation is discussed further in Section III.1.h.(1).(a).

The following pointer related restriction in the J73 manual precludes the use table types. "Items in tables declared with a <table-type-name> can only be accessed using pointers to the tables." Since pointers cannot be translated, tables declared with a type name cannot either.

(1) Table Dimension Lists

Ada requires both an upper and lower bound to appear in a range. J73 allows a default for the lower bound of 0. This will have to be explicitly supplied by a translator.

Ada requires that a range have a type_mark so that it will be specified as a particular discrete type. The J73 "*" dimension does not require a type. The J73 manual states: "(Note that in accordance

with Section 6.3.9 and 6.1, a bound of * dimensions range from 0 to NN-1, where NN is the number of elements in the corresponding dimension of the actual parameter, regardless of what the lower and upper bounds values are for the actual parameter or whether the bound has an integer or status type)."

(2) Table Structure

In J73, the programmer can specify the layout of tables in memory. Ada allows the programmer no control over the manner in which arrays of records are laid out.

(3) Ordinary Table Entries

There are a couple problems here. An equivalent to the J73 <order-directive> is not available in Ada.

J73 provides for 3 levels of packing, some in an implementation dependent fashion. Ada allows for one level through the PRAGMA(PACK). It is unclear whether the mapping of both (M)edium and (D)ense packing will have any effect on translated programs or not.

(4) Specified Table Entries

Specified table entries have corresponding record type representation constructs "use" and "at" (Ada LRM, Section 13.4). Some fairly complex reformatting of J73 representation specs will be required of the translator, but it can be done.

(c) Constant Declarations

Ada has no equivalent to the J73 concept of constant tables.

(d) Block Declarations

"A <block-declaration> declares a group of items, tables, and other blocks that are to be allocated in a contiguous area of storage." Presumably blocks are used to improve the access efficiency to data contained within the block. Ada does not define an equivalent construct. Perhaps this can be ignored during translation, but programmers who

specifically used a BLOCK construct probably did so for a reason. It is likely that some translations will be effected if block designations are ignored.

(e) Allocation of Data Objects

Ada does not explicitly provide a static allocation specifier. Variables contained in packages do remain allocated for the life of the package in which they are contained. Thus, data objects declared in packages are essentially static. One approach towards the translation of STATIC data might be to encapsulate all modules that define STATIC data inside a new package. The proliferation of packages each containing just one module for the sole purpose of achieving STATIC data would have a tremendous impact on the readability of programs. This is not the intended purpose of packages.

If a program requires STATIC data, the package construct must be used. Hand design of these packages is required to insure the quality of the resulting code.

(f) Initialization of Data Objects

There are several restrictions on when and where item presets can be used in J73. Many of these restrictions are not present in Ada. This has a minor effect on the robustness and style of the resulting Ada program.

J73 has preset lists. Ada has aggregates. Both languages have "equivalent" shortcut methods for representing repetitive values. J73 allows values within the preset list to be omitted. Ada requires a complete set. A translator can select an arbitrary literal of the appropriate type to complete the aggregate. This should have no impact on correctness. J73 requires assignment before use. This rule guarantees that the selected value will be replaced by a subsequent assignment before it is used.

This section mentions that preset values must be "implicitly convertible to the type of the data object being initialized." Implicit type conversions are are discussed in Section III.1.g.

(2) Type Declarations

Like options will have to be expanded before translation. The mechanics of saving textual information about previously defined types and substituting it for like options are difficult but not impossible.

(3) Statement Name Declarations

Ada does not allow statement names (labels) to be declared or passed as parameters. Labels used to name statements can be translated with no problem.

(4) Define Declarations

Ada has nothing equivalent to J73 DEFINE declarations. The concept of generics in Ada is close but does not have the same semantics. DEFINEs can be expanded before they are run through the translator resulting in a correct program, but the modularity and structure of the DEFINEs will have been destroyed. This will impact readability to the extent that DEFINEs are used in the original program.

(5) External Declarations

The Ada mechanism for exporting and importing name references is the package construct coupled with the WITH clause. Several compatibility problems exist between the J73 DEF - REF mechanism and the Ada package / WITH:

- Single names can be pulled out of compools through use of the DEF - REF mechanism. Ada has no such mechanism. The WITH clause imports all names declared within the referenced package. If REF specs are simply translated into WITH clauses with the compool/package name, some name conflicts may arise. Since Ada allows name overloading in some cases, the error may not be immediately apparent. In fact, REFerence to single names were likely made to avoid conflict with other names in the compool.

- For DEF specs that are not contained in compools, there is no corresponding Ada mechanism. The variable could be encapsulated within a package and then WITHed into the declaring module and all modules with a REF spec, but this is terribly cumbersome and results in poorly structured code. This further proliferation of packages should be avoided.

(6) Overlay Declarations

J73 allows entire objects or portions of objects (i.e. tables) to occupy the same storage space. The J73 manual states: "2) that certain objects are to occupy the same memory locations as other data objects." Ada strictly forbids the overlays. Section 13.5 of the Ada LRM states: "Address clauses should not be used to achieve overlays of objects or overlays of program units. Nor should a given interrupt be linked to more than one entry. Any program using address clauses to that effect is erroneous."

(c) Procedures and Functions

The syntax of J73 and Ada subprograms differ only slightly. There are several major semantic problems, however, primarily concerned with the definition of parameters.

(1) Procedures

All Ada subprograms can be called recursively and are reentrant. The REC and RENT <subroutine-attributes> can be ignored during translation.

(2) Functions

J73 uses the function name to store the return value of the function. The J73 manual states: "... the most recent value assigned to the <function-name> is used as the value of the function." This value cannot be subsequently used in a formula (expression), however. The use of a function name in a formula implies a call (perhaps recursively) to that function.

Ada does not use this same system. The return statement explicitly identifies the value or name of the value to be returned. In the general case, a translation would have to create a temporary variable with a generated name for use in storing the return value into. This will probably not impact efficiency since the value would have to be stored somewhere in the J73 program anyway. Problems with name generation are discussed in Section III.1.h.(2).(a).

(3) Parameters of Procedures and Functions

There are several problems in translating parameters from J73 to Ada. They are summarized below:

- J73 out parameters are equivalent to Ada in out parameters and should be translated as such.

- The colon between input and output parameters in J73 can be ignored during translation.

- Ada does not allow statement names or subprogram names as parameters. GOTOs to statement name parameters will be discussed in Section III.1.d.(7).

- J73 allows the programmers to designate the actual binding mechanism to be used during parameter passing. Ada provides no such capability, allowing the compiler to make the appropriate choice. In fact, the Ada LRM states: "A program is erroneous if its effect depends on which mechanism is selected by the implementation." J73 programmers who do specify the type of binding mechanism will have done so for a reason and will likely rely on the mechanism for the correct functioning of their program. This conflict in definitions cannot be resolved. Therefore, any subprograms that specify the parameter binding mechanism cannot be automatically translated. This is a serious violation of requirement 1.

- J73 and Ada differ significantly in their methods of defining of formal parameters. J73 allows the type definitions of

formal parameters to be given within the subroutine body. Type definitions may also be any type definition. Ada requires that formal parameters be given an immediate subtype indication. This means that the formal parameter must be declared as a subtype of some previously declared and visible type name. (Formal parameters cannot be directly declared as arrays or records or as enumeration types.) In order to be translated from J73 to Ada, the type definitions must be elevated to a level where both the subprogram definition and the module containing the subprogram call can see them. This elevation not only complicates the structure of the program but can also cause name conflicts.

(4) Inline Procedures and Functions

The PRAGMA INLINE could be used for translation. The J73 and Ada definitions seem compatible.

(5) Machine Specific Procedures and Functions

Any implementation or machine dependent functions provided within the language have no guarantee of having a correct translation.

(d) Statements

(1) Assignment Statements

- In Ada, the assignment operator is ":=" not "=".
- J73 assignments to a variable list will have to be expanded before translation. In order to avoid evaluating the right hand side of the assignment each time, assignments to subsequent variables should be made from either a temporary variable or the first variable assigned from the list. A temporary variable will require a name to be generated for it.

(2) Loop Statements

J73 and Ada differ in the definition of loop statements:

- Simple WHILE and FOR statements translate easily.
- Ada allows incrementation through a scalar range only by 1. Thus, J73 BY and THEN statements will have to be fabricated. An expression to calculate the correct value at each iteration must be formed as dictated by the original BY or THEN formula. A temporary variable to hold this value will also be required and all references to the original loop variable will have to be changed to reference the temporary variable. A <while-phrase> attached to a BY or THEN phrase will have to appear as an explicit test and EXIT. All this may require extra storage and additional computations though the impact should be minimal.
- A major problem occurs when the <control-item> in a J73 loop is a <control-variable> (is declared as a variable in the local scope). J73 allows modifications to such variables within the loop and use of their value after the loop statements is terminated. Ada does not allow this. Temporary variables will not work in this case. One might try to assign the value of the loop parameter to the variable declared in the outer scope just before exit from the loop. But the generalized GOTO will prevent a guarantee that the assignment will happen in all cases. Loop statements that have <control-variables> cannot be translated.

(3) IF Statements

Ada provides an "elsif" clause to allow for additional conditional tests before the final alternative "else". Translations from J73 will not take advantage of this feature (requirement 6).

(4) CASE Statements

The one problem with the CASE statement is the FALLTHRU clause. Ada does not have an equivalent construct. A correct

translation can be constructed by copying the executable statements from the following <case-alternative> into the previous statement list. Of course, this must be done in a "bottom up fashion" as there may be multiple FALLTHRUs. There are two possible problems with this. Copying can become excessive, resulting in messy redundant code. If the Ada compiler that will compile the resulting code is not able to notice that the copied code sequences are identical, additional machine code may be unnecessarily generated. The extent of the additional code is proportional to the number of FALLTHRUs in the original code.

(5) Procedure Call Statements

The colon between input and output parameters can be ignored during translation.

(6) RETURN Statements

Return statements within functions will have to be modified to return a value as discussed in Section III.1.c.(2).

(7) GOTO Statements

Ada does not allow labels as parameters or allow GOTOs to reference labels outside the scope of the GOTO statement itself. This type of GOTO cannot be translated. This J73 feature might be comparable to the Ada exception facility. Labels passed as parameters could designate "handlers" for errors within the subroutine. GOTOs to these labels could act as the Ada RAISE statement. Although a GOTO to a label passed by calling procedure may be used in this way, it can also be used in other ways that do not map into the Ada exception facility. It is safe to say that GOTOs to labels cannot be translated and that the Ada exception facility will not be used by a translation system.

J73 does not allow GOTOs into statements within a loop refered to in the manual as a <controlled-statement>. The manual does not prohibit GOTOs into a <conditional-statement> or into IF statements. Ada does not allow this. Such "out of scope" analysis will have to be performed prior to the translation of a GOTO.

(8) EXIT Statements

Ada EXIT statements are more powerful than those in J73 in that they can contain a WHEN clause.

(9) STOP Statements

Stop statements have no parallel in Ada.

(10) ABORT Statements

Abort statements, similar to GOTOs to statement names passed as parameters, cannot be translated into Ada. They are in no way equivalent to the Ada abort statement that relates to the Ada tasking facility.

(e) Formulas

It is unclear whether <compile-time-formula> functions are available in Ada. There are similar attribute functions available for some Ada types, but the J73 manual says that: "LBOUND, FIRST, and LAST are available reguardless of their arguments." The availability of functions such as NEXT, BIT, BYTE, SHIFTL, and SHIFTR will be discussed in Section III.1.f.(3).

(1) Numeric Formulas

The definitions for all numeric formulas are incomplete in that they do not specify what happens for error conditions. For example, the J73 manual specifies that: "The right operand of / and MOD must be non-zero." But it does not say what happens when it is zero. Range constraints are also specified but nothing is defined when they are violated. This is a serious semantic difference between J73 and Ada. An Ada exception is defined for all possible violations of language restrictions.

(a) Integer Formulas

The modulus operator is defined differently in J73 and Ada. Section 4.5.5 of the Ada LRM gives this definition for modulus:

A mod B = (A + K*B) mod B

J73 gives the following:

A mod B = A - (A/B) * B

These definitions do not give the same answer when A is negative and B is positive. Therefore, the MOD operator cannot be directly translated.

Ada also defines a REM (remainder operator). J73 does not.

(2) Bit Formulas

The only logical operator that Ada does not provide on BIT_STRINGs is the NOT operator. J73 does provide this. It is unclear whether the NOT operator can be composed from the operators provided in Ada and also be efficient.

(a) Relational Expressions

The definition of the relational operators relies heavily on the J73 definition of type compatibility for conversions. This will be discussed in Section III.1.g.

(b) Boolean Formulas

Ada defines additional short circuit forms for logical operators.

(f) Data References

(1) Variables

There are two problems concerning the translation of data references -- one major, the other minor. The minor problem concerns the name referencing of items within tables. J73 allows the name of the internal item to be used directly. Ada requires use of the dot qualifier to reference internal variables. All names referencing items declared within tables must be reconstructed by adding the outer scope name. This will complicate the translator.

A more serious problem already discussed is the definition of pointers in J73. Pointers cannot be translated directly. A human translator may even have to drop into assembly code to implement certain pointer properties unless a larger scope redesign can be found.

(2) Named Constants

(3) Function Calls

User defined function calls have no trouble being translated at the name reference level if they can be seen. Intrinsic function calls have more problems.

The LOC and NEXT functions are excellent examples of why pointers are not equivalent to access types. The NEXT function cannot be applied to enumeration types either unless the <next-argument> has a constant value of 1. Enumeration types do have the SUCC attribute but it does not take an argument.

The BIT function looks similar to an array slice operation on BIT_STRINGS. Right justification and padding with zeros, however, shows that they are not equivalent. The BYTE and SHIFT functions do not have Ada equivalents, but they could be included explicitly in a TRANSLATION_PACKAGE of sorts and be made visible to the whole program. The NEXT function on status types could also be implemented in this package.

The ABS function has an Ada equivalent. The SIGN function can be easily translated with relational operators. The BOUNDS functions LBOUND and UBOUND can be translated into the array attribute functions FIRST and LAST. The Status Inverse Functions can be translated in a similar way.

The SIZE and NWDSEN functions have some equivalents in Ada, although not all J73 variations are accepted. Length specification control is also available to assign the amount of storage in bits to be used for the representation of a certain type. Functions that return a size are not available in Ada.

(g) Type Matching and Conversions

In general, the rules governing type conversions are much less restrictive than they are in Ada.

Ada allows (explicit) type conversions in three cases. The following is a summary of the rules for allowed conversions. Complete definitions appear in Section 4.6 of the Ada LRM.

1. Numeric types can be converted to other numerics types. Conversions from a real value into an integer type involves rounding.

2. Conversion is allowed when the type of the operand is directly derived from the type mark of the conversion.

3. Array types can be converted when both the operand type and the type mark of the conversion have the same index and component types.

J73 allows many other legal conversions. The following is a list of incompatibilities:

- All conversions must be explicit in Ada. A type_mark is used to indicate the desired result type. Any implicit J73 conversions that are also legal in Ada must be given an explicit type_mark for conversion.

- Numeric conversion seems to be ok. Questions of accuracy are still unclear.

- By allowing any data object to be converted into a bit string and any bit string to be converted back to any other type, J73 completely destroys the concept of information hiding and data consistency. This capability allows anyone to access the "guts" of any data objects. Thus everything is available to anyone who can see it. This conflicts with one of the basic design tenets of Ada.

- The allowance for converting pointer types to integers and bit strings is a primary reason why pointers cannot be translated into access types.

- BIT_STRINGs are implemented as an array of BOOLEANs in Ada, and as such are governed by the rules for array conversion in Ada. J73 implicit conversions between bit strings of different sizes can therefore not be translated.

- Character strings are also implemented as arrays in Ada, along with the appropriate restrictions on conversion.

(h) Basic Elements

(1) Characters

In Ada, the predefined enumeration type CHARACTER is provided in the STANDARD package defined in appendix C of the Ada LRM. MIL-STD-1589B states: "Each implementation must define these characters, as well as the ordering of all <characters> in a collating sequence." The fact that J73 character ordering is implementation defined is incompatible with Ada.

(2) Symbols

(a) Names

As discussed in previous sections, there are instances when the translator will have to generate a name. To prevent conflict with other names declared within the same scope, the name must be unique. Identifier names should also be readable and imply something about the object which they denote. The combined requirements of uniqueness and readability are incompatible. In order to guarantee uniqueness, readable names cannot be used; they are likely to already exist. One possibility is to use a character allowed in Ada but not in J73 such as the underscore character. Names with an underscore anywhere would always be unique as long as the translator did not generate the same name twice. The key problems with this is making the generated name make sense in the local context. This requires a handle name already declared in the local scope and the attachment of and underscore and a suffix or prefix. Even this does not guarantee a suitable name. The best solution is to generate a definitely unique name and do the best possible with its actual content.

Ada allows the use of the underscore character in identifiers to make them more useful. Jovial does not allow this. Thus, since identifier names are translated verbatim, they will not appear in the same style as Ada identifiers. This conflicts with requirement 4.

Jovial allows the use of dollar sign characters in identifiers and claims that they are "translated to an implementation-dependent representation". This is incompatible with Ada.

(b) Reserved Words

Any name that is not in the J73 reserved word list can be used as an identifier in a J73 program. There are, however, some Ada reserved words that do not appear in the J73 reserved list. They are:

ACCEPT	ACCESS	ALL	ARRAY	AT
BODY	DECLARE	DELAY	DELTA	DIGITS
DO	ELSIF	ENTRY	EXCEPTION	FUNCTION
GENERIC	IS	LIMITED	LOOP	OF
OR	OTHERS	OUT	PACKAGE	PRAGMA
PRIVATE	PROCEDURE	RAISE	RANGE	RECORD
REM	RENAMES	REVERSE	SELECT	SEPARATE
SUBTYPE	TASK	TERMINATS	USE	WHEN

This poses the potential for conflict.

(3) Literals

Ada has the concept of enumeration literals. J73 does not have a corresponding status literal.

(a) Numeric Literals

Some numeric literals may have to be slightly reformatted.

(b) Bit Literals

J73 bit literals will have to be converted into BIT_STRING aggregates.

(c) Boolean Literals

J73 1B'1' and 1B'0' will have to be converted into the TRUE and FALSE literals.

(d) Character Literals

J73 character literals will have to be converted into Ada character and string aggregates.

(4) Comments

Although seemingly innocuous, comments pose a very serious problem. The syntax translation from the J73 "comment" or %comment% to the Ada --comment is obviously trivial. But the translation of the actual wording of the comments themselves is not.

Comments often refer to language constructs. Comments in a J73 program might read: "This table is used to store aircraft attitude vectors." Or: "Value-result binding is used here to ...." If these comments were to be translated verbatim, they would be confusing and self defeating. Comments may also refer to names which have disappeared during translation. The names of DEFINE constructs which have been expanded during translation is an example. J73 numeric type designators are another.

A complicating factor is our ability to recognize when comments are relevant and helpful and when they are not. Unless a translating system is prepared to solve the problem of deciphering the English language, it can safely be said that all comments must be suspect and therefore discarded. This is extremely damaging to the quality of the resulting code. Of course, a human could run through the code and fill in comments by looking at the original code, but this would substantially increase the percentage of work required after translation.

(i) Directives

The J73 manual states: "<Directives> are used to provide supplemental information to a compiler about the <complete-program>, and to provide compiler control." This makes them comparable to Ada pragmas. Some of the predefined J73 directives match well with Ada predefined

pragmas. These are primarily text and listing control directives such as COPY, SKIP, BEGIN, and END.

Some J73 directives violate the Ada language definition such as expression evaluation directives, initialization directives, and allocation order directives. The use of the !LEFTRIGHT directive in a J73 program has very serious consequences. This directive forces left to right evaluation of operators at the same precedence level. This is incompatible with the Ada LRM statement that "A program that relies on a specific order (for example because of mutual side effects) is therefore erroneous." The reason that this is so serious is that Ada programs that contain dependencies in the evaluation order of operands will compile without error, but may not execute as intended. All code within the directive !LEFTRIGHT must therefore be suspect and can not be guaranteed to be semantically equivalent.

Other directives do not violate Ada but are unlikely to be included in the Ada compiler on which the resulting code must be compiled. These include linkage directives, trace directives, reducible directives, and register directives.

As mentioned in Section III.1.a.(2).(a), compool directives can be translated into Ada WITH clauses. This is not entirely true. J73 allows any name declared within the compool to be directly referenced by a compool directive. Ada allows reference only to package a subprogram modules that appear as library units. A reference to a particular item or table in a compool was probably made to avoid a name conflict with some other name in the compool. This problem is similar to the REF - DEF problem described in Section III.1.b.(5).

2. SUMMARY OF UNTRANSLATABLE FEATURES

J73 and Ada have a variety of incompatibilities. There are several basic design tenets of each language that do not match well. This results both in constructs that have no equivalent in Ada and ones for which a correct translation has a major impact on the quality of the resulting software. Still other J73 features are considered to have high risk for translation. These features have definitions that are very similar to Ada, but anomalies in their implementations may result in some incompatibilities in some translations. These classifications are summarized in Figure 1.

Conflicting Design Concepts:
- Information Hiding
- Type Composition
- Type Conversion
- Data Representation and Access
- Name Importation/Exportation
- Error Handling

Specific Untranslatable Constructs:

Declarations:
- Pointers
- Table Structure Specifiers
- Statement Name Declarations

Procedures and Functions:
- Formal Parameter Declarations
- Machine Specific Procedures

Statements:
- LOOPs with Control Variables
- GOTOs to Statement Names
- STOP and ABORT Statements

Type Conversions:
- Primarily Conversions to and from INTEGER and BIT Types

Directives:
- LEFTRIGHT Directive
- Some COMPOOL Directives

Translations Impacting Quality:
- Static Allocation
- Define Declarations
- External Declarations
- LOOP temporary variables
- CASE FALLTHRU option
- Name Generation
- Comments

High Risk Constructs:
- Numeric precision
- Numeric truncation and rounding
- Blocks
- Bit string operators

Figure 1. J73 - Ada Definition Conflicts

3. PERCENTAGE TRANSLATABLE

The percentage of J73 constructs that can be automatically translated into Ada can be measured in two ways. The first method is a straight ratio between those constructs that can be translated and those that cannot. This measure has limited utility, however, since our goal is to translate real J73 programs, and not just the reference manual. A more useful metric is the average percentage of real J73 programs that can be translated. This measure takes into consideration the relative frequency of constructs appearing in real programs. It also considers the amount of local translatable code that is "poisoned" by constructs that cannot be translated.

It is very difficult to estimate how much code will be poisoned by other local untranslatable statements. This can happen in several ways. 1) The construct may be an integral part of the local algorithm. Even though most of the algorithm can be translated, the lack of the untranslatable construct will likely prohibit the module from performing its assigned task. It is also unlikely that there is a quick, local patch. If there were, the translator would be able to substitute it as an equivalent construct. 2) Illegal declarations can invalidate references to those objects. 3) The LEFTRIGHT directive is very pervasive. Any code within the area affected by this directive must be suspect. 4) DEFINEs are heavily used in J73 programs. If their negative impact on program modularity cannot be tolerated, large chunks of code will not be translated. 5) GOTOs into IF statements or to parameters. It is safe to say that most J73 constructs that violate the rules of Ada will poison much of the surrounding code.

What then is the average percentage translatable? With the above discussions in mind, 30% to 40% of all J73 programs should be achievable with a good system.

4. SUMMARY OF UNUSED ADA CONSTRUCTS

The following Ada features have no equivalent J73 constructs and will therefore be absent entirely from automatically translated programs. Programs translated into Ada will use a subset which does not include

these features. (If translation is augmented by human translation, some of these features may be used.)

- Tasking Facility.
- Exception Handling Facility.
- Generics.
- Ada I/O.
- Access types; dynamic data allocation.
- Overloading

The following Ada features are not utilized to their fullest potential due to restrictions in J73.

- Packages - Private types, variable declaration.
- Typing system - general type composition, subtyping, some type attributes, discriminant records, array slice operations.

5. COST EFFECTIVENESS

The stated objective was to remove the need for maintaining a J73 programming environment by switching all code into Ada; thereby removing the cost of maintaining it. These cost savings must be weighed against the cost of developing a translation system, the cost of translating large amounts of complex software, and the differential cost, if any, between maintaining the program in the J73 and Ada environments. This section will not attempt to attach actual figures to each cost but will outline the types of costs that can be expected. Estimates will be given when known.

a. Translation System Development Costs

As stated in Section III.2.a., the complexity and thus the cost of a translation system would be similar to the cost of a compiler. It is unclear, however, whether just one translator can handle all J73 translations. The analysis in Section III.1 provides several examples where MIL-STD-1589B is ambiguous and contains many implementation dependent features. Several interpretations of MIL-STD-1589B exist and are embodied in J73 compilers used today. Programs that work correctly

when compiled on these systems will require the same interpretation set in the translator in order to be translated correctly. Each point of interpretation must be reflected in a translator option in order to provide a correct interpretation and translation.

A translation system would be a short-lived system. Once all J73 was translated into Ada, the system would have not further use. Thus, it would not require the normal maintenance to fix bugs. This is a blessing in disguise, however. It means that all (or an extemely high percentage) bugs must be removed before it can be successfully used at all.

b. Code Translation Costs

Once a translator is built and functioning correctly, the primary cost will be the labor of programmers skilled in both J73 and Ada. They would be required to clean up the translation to provide a full translation. These cost are directly proportional to the amount of human translation required.

Of course, the resulting translated code must be entirely retested to certify that the new program satisfies all of the functional requirements. This is very often non-trivial, expensive operation. At this stage the program could be considered an Ada program and all modifications made in Ada.

c. J73 versus Ada

Ada was designed to reduce the cost of maintaining software through the use of new concepts in the structuring of programs and data. These concepts were not placed in J73. As we have seen in Section III.1, the features in Ada that were designed for this purpose could not be correctly utilized by a translator. We, therefore, cannot expect to realize the advantages of Ada. We can expect the resulting programs to require the same effort to maintain as the originals.

SECTION IV

CONCLUSIONS AND RECOMMENDATIONS

Several conclusions can be drawn from the discussion above:

- A high percentage of real J73 programs cannot be automatically translated.
- The translation of numerics is risky. Some precision errors must be anticipated in some translations.
- A large amount of human supplement is required to fully translate J73 programs into Ada.
- The resulting code may be poorly structured. Several of the program structuring facilities in J73 and Ada have incompatibilities. Several Ada structuring facilities will not be utilized by a translator.
- The resulting code may be hard to read and understand. The lack of translatable comments constitutes the largest impact. Name translation and name generation also effect readability.
- The style of the resulting program will still be J73 style. A translator will only rewrap J73 style programs in Ada syntax.
- The resulting programs will not be as robust as they should be. Several Ada features are not used by the translator.

Due to the overwhelming number of negative conclusions development of a J73 to Ada translation system is not recommended at this time. The best solution to the problems is to leave the J73 programs as they are until their life cycle is terminated. If programs must be translated into Ada, it is recommended that they be redesigned in Ada and entirely hand translated into Ada.

One possible use for our capability to translate some J73 constructs might be the development of a "local" translator. A human could bracket off portions of code that can be translated effectively. If some portion

of the bracketed code could not be translated, then no part would be translated. These segments of translation could be used in conjunction with an editor to hand translate programs. Such a "local" translator could remove the tedium of translating these portions of code.

☆ U.S.Government Printing Office: 1983 — 759-062/542

END

FILMED

12-83

DTIC

(i) Directives

The J73 manual states: "<Directives> are used to provide supplemental information to a compiler about the <complete-program>, and to provide compiler control." This makes them comparable to Ada pragmas. Some of the predefined J73 directives match well with Ada predefined

correct translation has a major impact on the quality of the resulting software. Still other J73 features are considered to have high risk for translation. These features have definitions that are very similar to Ada, but anomalies in their implementations may result in some incompatibilities in some translations. These classifications are summarized in Figure 1.

Figure 1. J73 - Ada Definition Conflicts

4. SUMMARY OF UNUSED ADA CONSTRUCTS

The following Ada features have no equivalent J73 constructs and will therefore be absent entirely from automatically translated programs. Programs translated into Ada will use a subset which does not include